TP01 - Estrutura de dados

Marcos Daniel Souza Netto - 2022069492

October 15, 2023

1 Introdução

O problema proposto foi dividido em duas partes: primeira, a implementação de um solucionador de equações booleanas dado o valor das variáveis; segunda, a implementação de um solucionador SAT (Verificar se a esquação é satisfazível) com a restrição de no máximo cinco quantificadores.

2 Método

A seguir serão detalhados a implementação da soluções, de modo a explicitar as estruturas de dados utilizadas e as estratégias de solução. A priori vale apresentar as especificações do ambiente de desenvolvimento utilizado:

- Sistema Operacional: Ubuntu 22.04.3 LTS;
- Compilador: G++ 11.4.0;
- Processador: Ryzen 5 5500u;
- Memória RAM: 8GB.

2.1 Estruturas de dados

Em síntese, foram utilizadas as seguintes estruturas de dados:

- **Stack**: Representa uma pilha de elementos, sendo utilizada em diversos trechos do código, como por exemplo para percorrer a árvore de soluções e para transformar a expressão infixa (Na ordem habitual que estamos acostumados, como por exemplo $a \vee b$) para a forma posfixa ($ab\vee$);
- **Binary Tree**: Representa uma árvore binária, sendo utilizada para representar a árvore de soluções na parte 2 do trabalho;

A implementação dessas estruturas se localizam nos arquivos *stack.h* e *binary_tree.h*.

2.2 Classes e principais funções

O código foi dividido em três grandes classes, sendo elas:

- **stack.h**: Implementação da estrutura de dados pilha, e contém as principais operações de uma pilha, como *push*, *pop*, *top*, *isEmpty* e *size*, que respectivamente realiza a inserção de um elemento, a remoção do elemento do topo, retorna o elemento do topo, verifica se a pilha está vazia e retorna o tamanho da pilha;
- **binary_tree.h**: Implementação da estrutura de dados árvore binária, e contém a definição de nó dessa árvore além das principais operações em árvore, como a inserção de filhos à esquerda ou à direita de um nó, e o caminhamento na árvore, que pode ser feito em pré-ordem, em ordem ou em pós-ordem;

- **utils.h e utils.cpp**: São respectivamente, definição e implementação das principais funções de manipulação das estruturas de dados utilizadas para a solução do problema.

  infixToPostfix(string infix): Recebe uma string contendo uma expressão booleana na forma infixa e retorna um Vetor de inteiros representando a mesma expressão na forma posfixa;

  evaluateExpression(Vector<int> postfix, int arr[]): Recebe representação posfixa da expressão na forma de vetor de inteiros e um vetor contendo os valores das variáveis da expressão(0 $\rightarrow$ false e 1 $\rightarrow$ true) e retorna o resultado da expressão(0 $\rightarrow$ false ou 1 $\rightarrow$ true);

  satTree(Vector<int> postfix, string vals): Recebe um Vetor de inteiros representando a expressão booleana na forma posfixa, um vetor contendo os valores das variáveis da expressão(0 $\rightarrow$ false, 1 $\rightarrow$ true, e $\rightarrow$ existe, a $\rightarrow$ para todo) e retorna 1 ou 0 caso a expressão seja satisfazível ou não, respectivamente. Caso ela seja satisfazível, uma string contendo a possível solução é retornada;

- **main.cpp**: Contém a função principal do programa, que realiza a leitura dos dados de entrada e chama as funções de solução do problema.

3 Análise de Complexidade

A seguir serão apresentadas as análises de complexidade das principais funções do programa.

3.1 infixToPosfix

Como mencionado acima, esta função converte uma expressão infixa para a representação posfixa. Para isso, a função percorre a string contendo a expressão infixa, e para cada caractere, verifica se ele é um operando ou um operador. Caso seja um operando, ele é adicionado à string de saída, caso seja um operador, ele é adicionado à pilha. Ao final da leitura da string, todos os operadores que estão na pilha são adicionados à string de saída. Portanto, a complexidade da função é $O(n)$, onde n é o tamanho da string de entrada.

A complexidade de espaço é $O(n)$, pois a pilha pode conter no máximo n elementos.

3.2 evaluateExpression

Esta função recebe um vetor de inteiros representando a expressão booleana na forma posfixa e um vetor contendo os valores das variáveis da expressão e retorna o resultado da expressão. Para isso, a função percorre esse vetor de entrada, e para cada elemento, verifica se ele é um operando ou um operador (inteiros negativos). Caso seja um operando, ele é adicionado à pilha, caso seja um operador, ele é removido dois operandos do topo da pilha ou apenas um operando (No caso do operador $\sim$), e o resultado da operação é adicionado à pilha. Ao final da leitura da string, o resultado da expressão é o elemento que está no topo da pilha. Portanto, a complexidade da função é $O(n)$, onde n é o número de elementos do vetor de entrada.

A complexidade de espaço é $O(n)$, pois assim como a função anterior, a pilha pode conter no máximo n elementos.

3.3 satTree

Esta função recebe um vetor de inteiros representando uma expressão booleana na forma posfixa, um vetor contendo os valores das variáveis da expressão(agora podendo conter quantificadores existe e para todo) e retorna 1 ou 0 caso a expressão seja satisfazível ou não, respectivamente. Caso ela seja satisfazível, uma string contendo a possível solução é retornada.

Para isso, a função indexa os quantificadores da expressão, em seguida percorre o vetor dos quantificadores construindo a árvore de solução pelos últimos quantificadores até o primeiro. A árvore possui, portanto, como folhas os possíveis valores para os quantificadores (0 ou 1) e desse modo ela possui 2^k folhas, onde k é o número de quantificadores, seja existe ou para todo. Para exemplificar considere a figura 1.

```
1
2   Exemplo: $(EXE) -s "0 | 1 & 2 & 3" e11e
3
4   Vetor valores das variáveis:                                    e11e
5   Nós internos após a valoração da última variável:          e111        e110
6   Nós folhas:                                  1111    0111        1110    0110
7   Valores da expressão dada a valoração anterior:     (TRUE)  (TRUE)      (TRUE)  (FALSE)
8   Com base nos valores, montamos nossa string:               a111        1110
9   Resultado da expressão:                                         111a
10  |
11
```

Figure 1: Árvore de solução para a expressão $\exists a \exists d, b = c = 1, (a \vee b \wedge c \wedge d)$

Como visto, a árvore possui 2^k folhas, e para cada uma delas, a função *evaluatePostfix* é chamada para verificar se a expressão é satisfazível. Portanto, a complexidade da função é $O(2^k)$, onde k é o número de quantificadores.

Já a complexidade de espaço é $O(k)$, pois a solução da árvore é realizada de modo ir o mais profundo na árvore possível, e cada vez que um nó é resolvido, seus filhos são removidos da memória.

4 Estratégias de robustez

A estratégia de robustez utilizada foi basicamente a verificação da entrada do usuário, ou seja a verificação na quantidade de argumentos e seus tipos.

5 Análise Experimental

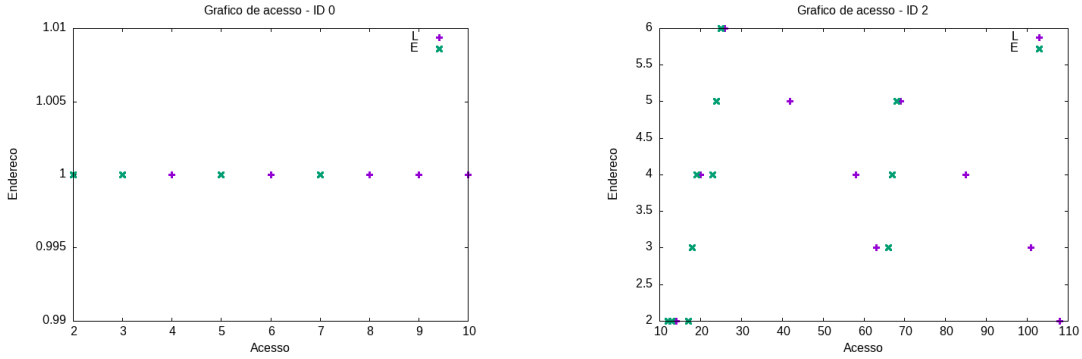
A seguir serão apresentados os resultados obtidos na experimentação do programa.

5.1 Localidade de Referência

Utilizando o programa analisamem, fornecido pelo professor, foi possível observar a localidade de referência do programa. Importante notar que aqui serão apresentadas imagens relacionadas a parte de satisfatibilidade do programa, afinal utiliza a parte de solucionador de equaações booleanas como subrotina.

Ademais, foi utilizada a seguinte expressão para a parte de satisfatibilidade:

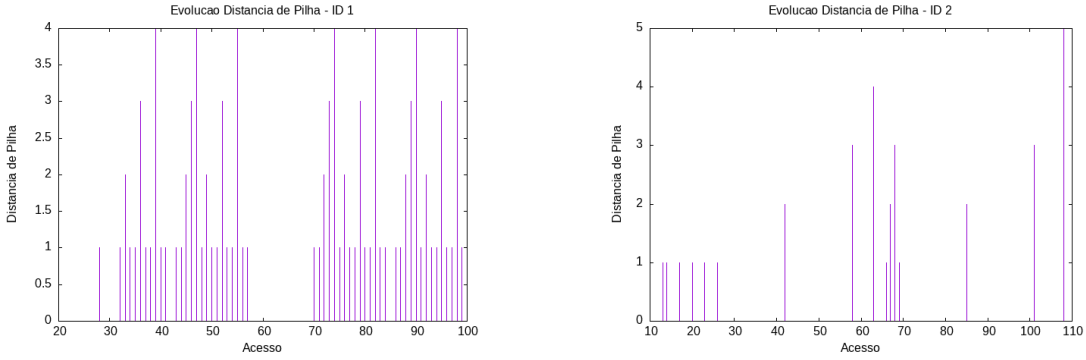
```
./bin/tp01 -s "0 | 1 & 2 & 3" e11e
```



(a) Pilha da função infixToPostfix

(b) Pilha da função sat_tree

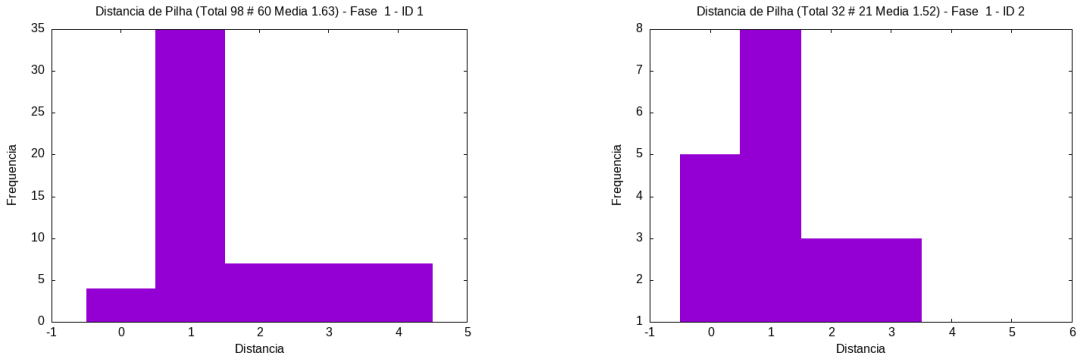
Figure 2: Acesso de Memória



(a) Pilha da função evaluateExpression (b) Pilha da função sat_tree

Figure 3: Evolução de distância da pilha

Observa-se nessa Figura 3 (a) que ocorre um padrão em 4 partes, isso se deve porque a função *evaluateExpression* é chamada uma vez para folha da árvore. Outro detalhe sobre a Figura 3 (b) é a possibilidade de observar que o caminhamento na árvore é realizada semelhante a DFS(*Depht First Search*), consonante com a implementação da função *sat_tree*.



(a) Pilha da função evaluateExpression (b) Pilha da função sat_tree

Figure 4: Distância da pilha

5.2 Tempo de execução

Nesse experimento foi realizado medições de tempo de execução a parte de satisfatibilidade do programa, variando o número de quantificadores da expressão. Para isso, variamos a quantidade de quantificadores da expressão de 13 até 20, para que possamos observar o comportamento do programa para um número maior de quantificadores. Teve como base o seguinte comando (Atente-se que estamos variando o número de quantificadores):

```
bin/tp01 -s "0 | 1 | 2 | (...) | 17 | 18 | 19" eeeeeeeeeeeeeeeeeeee
```

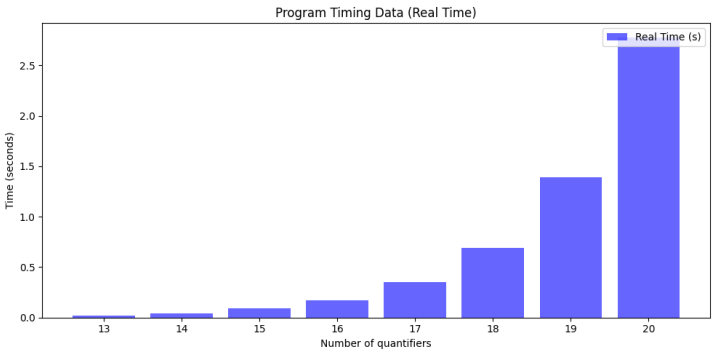


Figure 5: Tempo de execução do programa para diferentes números de quantificadores

Observa-se, portanto, comportamento semelhando a ordem de complexidade descrita anteriormente ($O(2^k)$), onde k é o número de quantificadores.

6 Conclusões

O trabalho proposto foi de grande valia para o aprendizado de estruturas de dados, e de como elas podem ser utilizadas para solucionar problemas, afinal foi de extrema importância a utilização de pilhas e árvores como estruturas principais.

Além disso, podemos observar que a análise de complexidade de grande importância, uma vez que ela nos permite prever o comportamento do programa para diferentes entradas, e assim, podemos realizar otimizações no código, como por exemplo, a utilização de uma estrutura de dados mais eficiente. Nesse trabalho, por exemplo, o problema de satisfatibilidade é um problema NP (determinístico não polinomial), ou seja, não existe uma solução conhecida em tempo polinomial, o que condiz com a solução implementada (de complexidade exponencial).

Por fim, a análise do tempo de execução de forma experimental foi de fato interessante pois permitiu conferir a similaridade com a teoria e o mundo real.

Bibliografia

Slides da disciplina de Estrutura de Dados, ministrada pelo Prof. Wagner Meira Jr. e Prof. Eder Fereira Figuiredo.

Instruções para compilação e execução

Em um terminal, navegue até a pasta raiz do projeto e execute os seguintes comandos:

```
$ make
$ ./bin/tp1.out -a "<expressão booleana>" <valores das variáveis> // Valor
$ ./bin/tp1.out -s "<expressão booleana>" <valores das variáveis> // SAT
```